Open in app

Parveen Khurana

125 Followers About Following

Long Short Term Memory(LSTM) and Gated Recurrent Units(GRU)

P Parveen Khurana Jul 17, 2019 · 14 min read

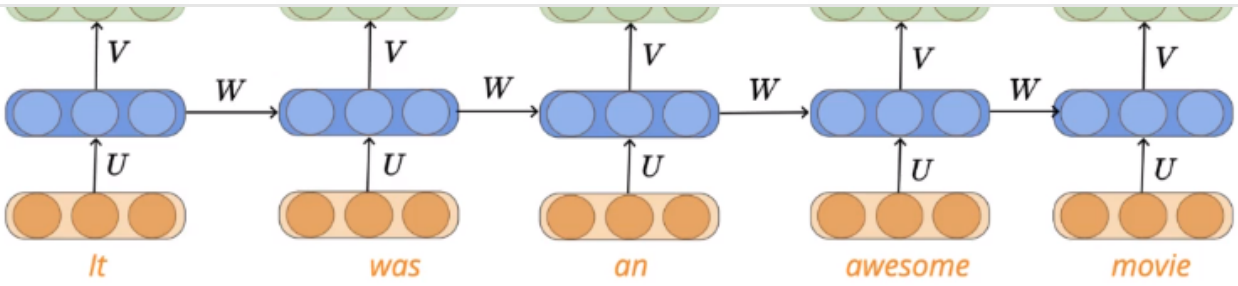
This article covers the content discussed in the LSTMs and GRU module of the Deep Learning course offered on the website: https://padhai.onefourthlabs.in

Dealing with Longer Sequences

The problem with the RNN is that we want the output at every time step to be dependent on the previous input and the way we do that is that we maintain a state vector(**st**) and we update it at every time step by making it a function of the current input and the previous state (**s(t-1)**). So, in a way, at every time step, the information stored in the blue vector(hidden state **st**) is getting updated as compared to the information in the previous hidden state as we are taking into account the current time step input, some weightage of previous hidden state and the bias term, and the information in the current time step hidden state is getting morphed as compared to the information in the previous time step hidden state.

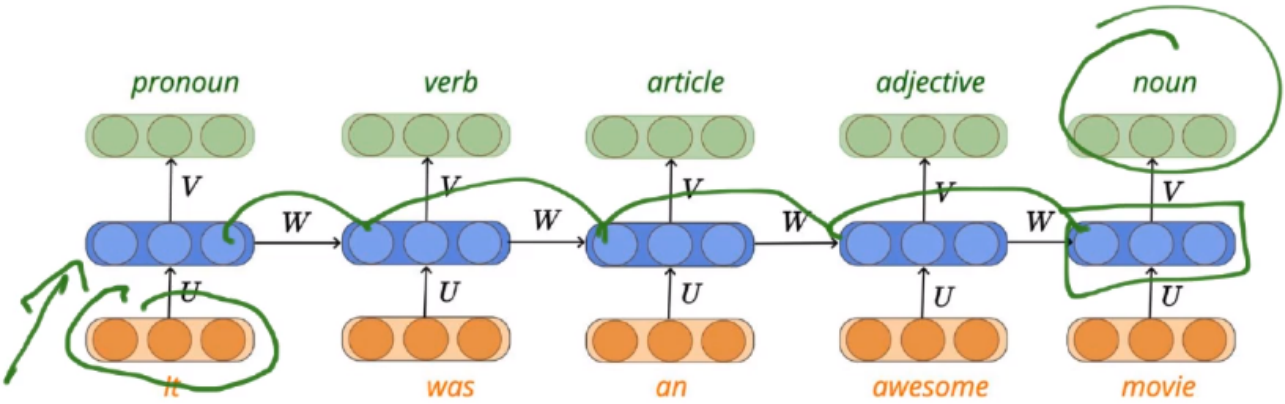
Now if we have a very long sequence say 15–20 words, then by the time we reach the last word and would like to do the classification, the information from the first hidden state would have been completely morphed and in some cases, this might affect the result drastically. So, this is the problem with the usage of RNNs for longer sequences and if we even try to deal with it and update the corresponding parameters using the backpropagation with time, then we run into the problem of Vanishing/exploding gradients.

Open in app



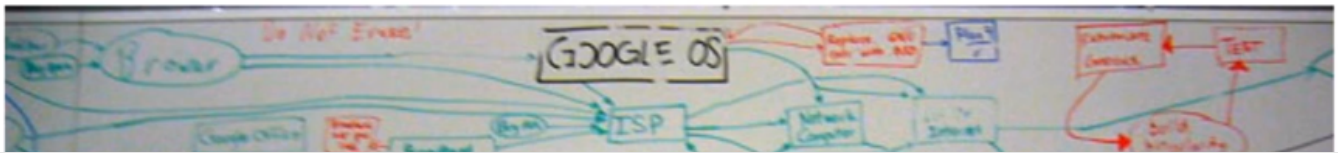
- At each new timestep the old information gets morphed by the current input
- One could imagine that after t steps the information stored at time step t − k (for some k < t) gets completely morphed

- Even during backpropagation the information does not flow well



The whiteboard analogy

The analogy to the above-described scenario is this whiteboard analogy. So, we have a whiteboard(of fixed width and height) and if we start writing information on it(say every 10–15 seconds we write some information on it), then after a while this whiteboard would become so messy and it would become very difficult to figure out what was the information that we wrote at time step 1 when we are at time step 100.

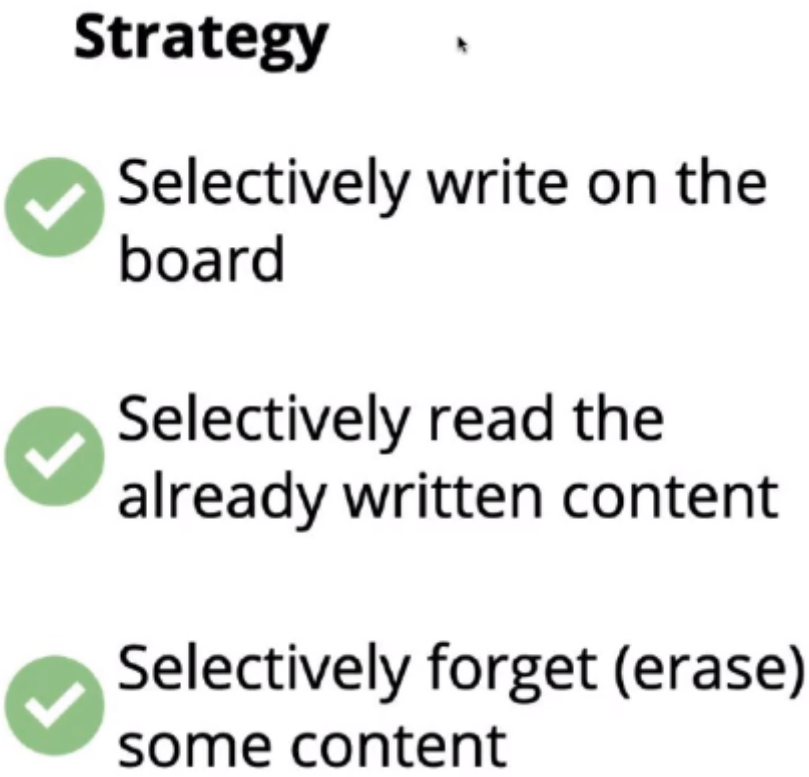


Open in app

The same problem we have here, this state vector (**st**) is of fixed size(‘d’ dimension), and we are writing information on to it at every time step. So, after a while, it would become very difficult to know what the first input contributed to this state vector (**st**)and this would become even more challenging in case of longer sequences.

The solution to this also comes from the whiteboard analogy.

Suppose we have a fixed size whiteboard and we want to solve something on it. We will follow the below strategy:



Open in app

$$a = 1 \quad b = 3 \quad c = 5 \quad d = 11$$

$$\textbf{Compute } ac(bd + a) + ad$$

The way we go about this computation is the following:

1. ac
2. bd
3. $bd + a$
4. $ac(bd + a)$
5. ad
6. $ac(bd + a) + ad$

So, there are these 6 steps but we cannot write all these 6 steps at a time on the whiteboard as the whiteboard has a small and fixed size. So, what we actually do here is that we compute the below mentioned 3 values(assuming that at a time whiteboard can accommodate only 3 statements):

$$ac = 5$$

$$bd = 33$$

$$bd + a = 34$$

Open in app

Now the next computation that we need to make is (ac*(bd + a)) that means we don't need (bd) anymore as it has already been used for computing (bd + a) and the next computation would be (ac*(bd + a)) for which we don't need the value of (bd) (so we will erase this from whiteboard, point 3 of the strategy, **selectively erase** some content) as long as we have the value of (bd + a), then we compute the value of (ac* (bd + a)) using the value of (ac) and (bd + a) which makes up the point 2 of the strategy ie. **selectively read**

$$ac = 5$$

$$ac(bd + a) = 170$$

$$bd + a = 34$$

Now again we have filled up the whiteboard and the next value that we need is of (ad), we don't need the value of (ac) and (bd + a) anymore so we can erase anyone or both of these two statements from the whiteboard.

$$ac = 5$$

$$ac(bd + a) = 170$$

$$ad = 11$$

We have written the value of ad directly as 11(and not like 1*11 = 11) which would have taken 2 steps, so this corresponds to point 1 of the strategy which says about

Open in app

$$ad + ac(bd + a) = 181$$
$$ac(bd + a) = 170$$
$$ad = 11$$

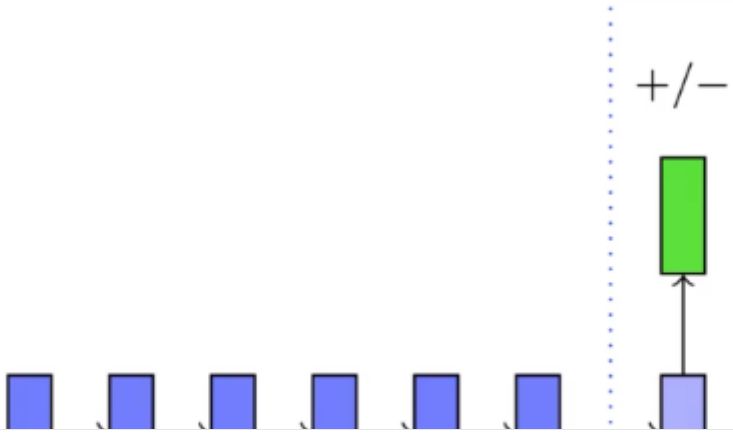
So, again we **selectively erase/forget** the value of 'ac' from the board and then **selectively write** the required value.

So, this is typically how we deal with the short memory or small memory or small dimension of a whiteboard which can only store so much information and the strategy that we use is that we selectively read, write and forget the content.

Can we use a similar strategy in RNNs?

Since the RNN also has a finite state size, we need to figure out a way to allow it to selectively read, write and forget

An example where RNNs need to selectively read, write and forget



Open in app

Review: The first half of the movie was dry but the second half really picked up pace. The lead actor delivered an amazing performance

If we look at this review, it started off with a slightly negative statement but from there on the information changes, it says 'the second half really picked up pace. The lead actor delivered an amazing performance'. So, we actually want to classify this review as positive.

We forget the information added by stop words(as called in Natural Language Processing(NLP), these are not adding any important information, they are not important to make the final decision)

Review: The first half of the movie was dry but the second half really picked up pace. The lead actor delivered an amazing performance

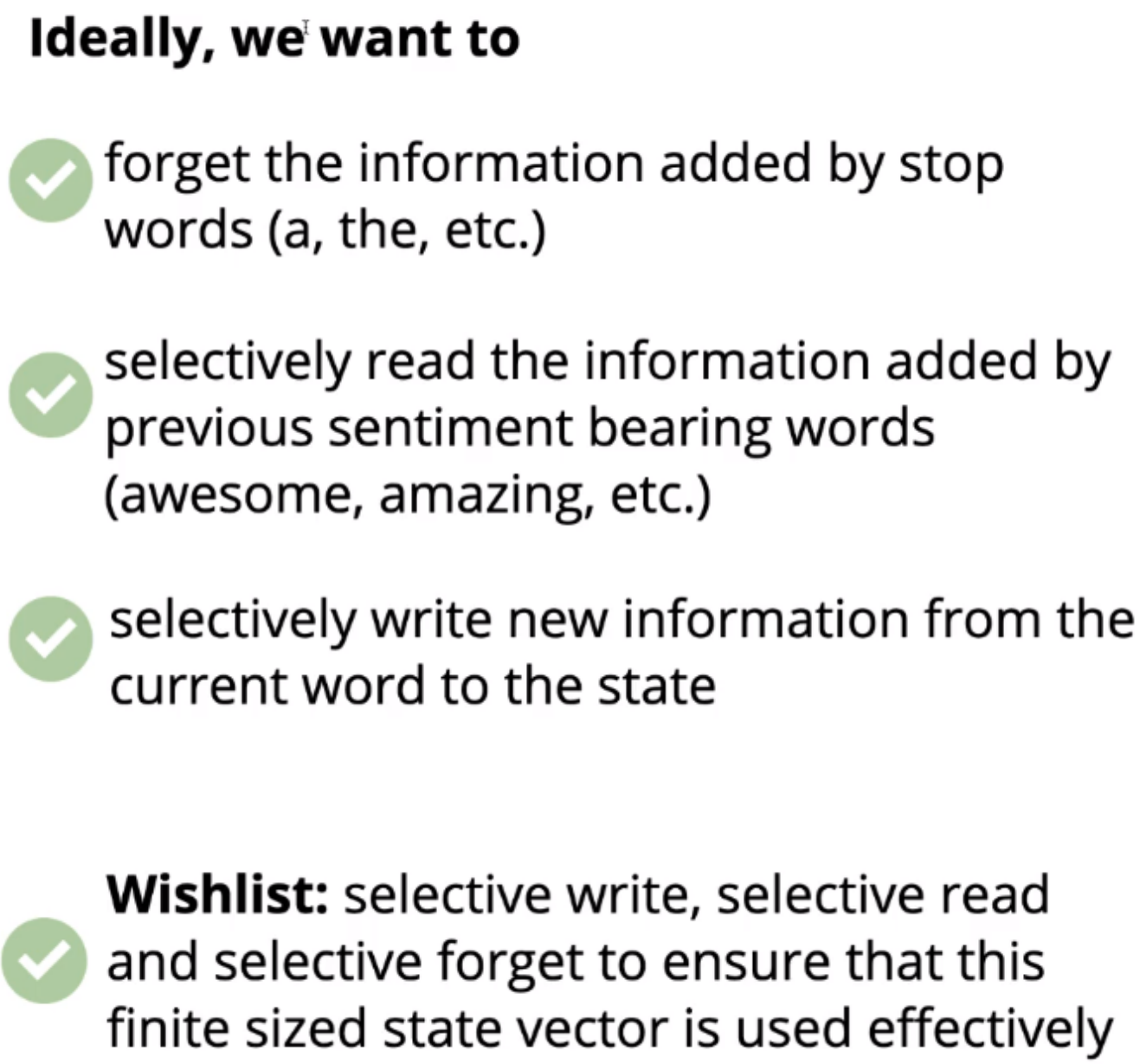
stop words in red underline

We selectively read the information added by the previous sentiment bearing words(example words like amazing, awesome, etc. which would add some good information for the prediction of the final class).

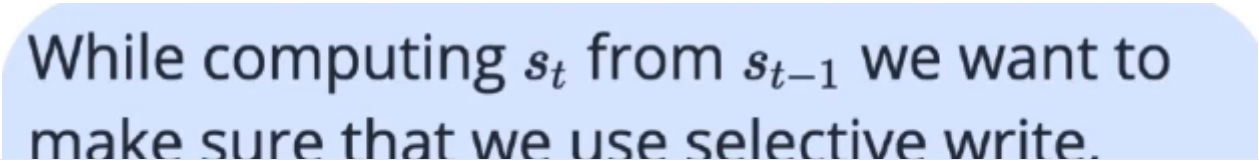
Review: The first half of the movie was dry but the second half really picked up pace. The lead actor delivered an amazing performance

Open in app

And then we selectively write the information from the current word to the state.

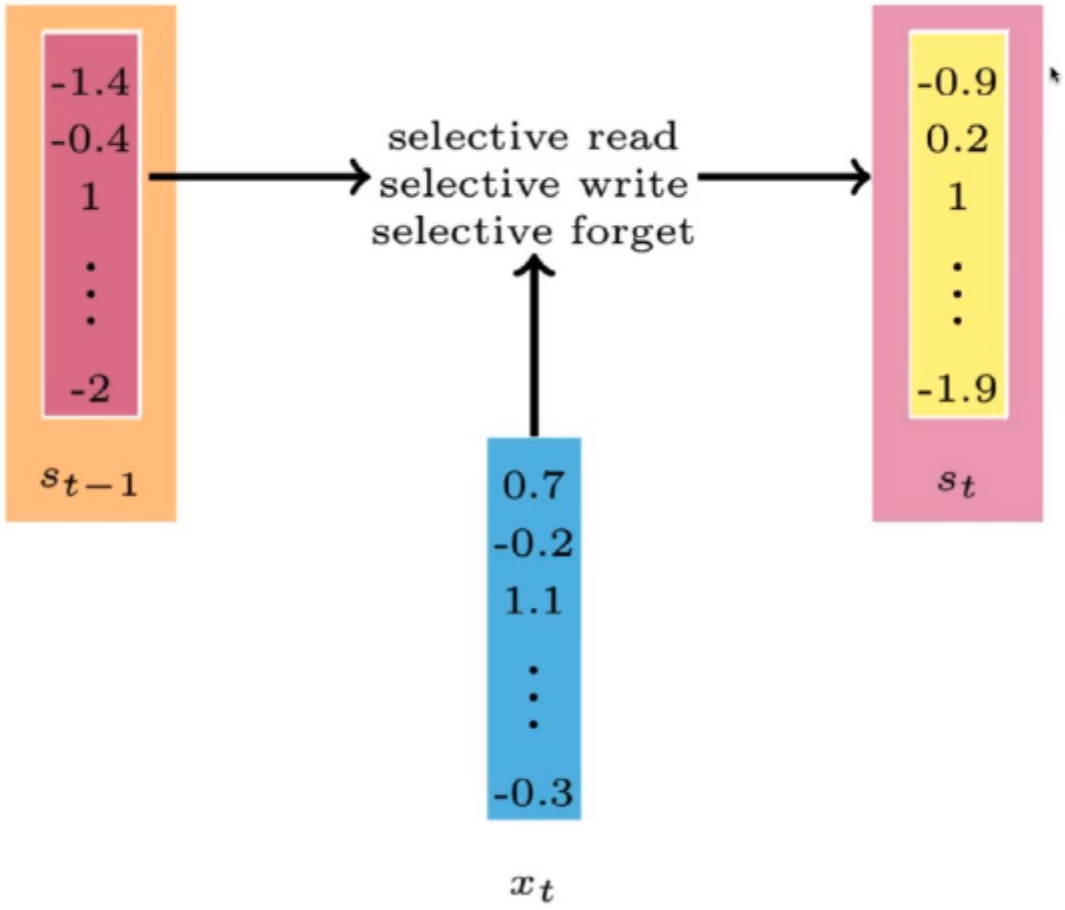


We compute the **st**(state at time step t) from the input **xt**(at time step t) and the from the state **s(t-1)** (state at time step (t-1)) and we want to make sure that we use the same analogy and try to use the selective read, write and forget so that this state at time step 't' (**st**) of finite size does not get over bin, it just stores the relevant information and keeps doing at every time step

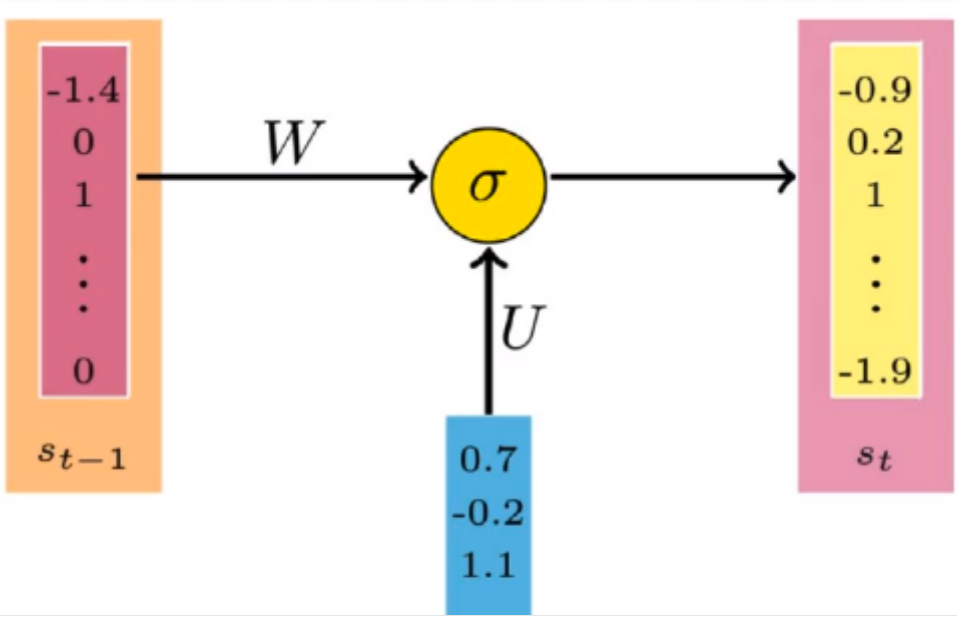


Open in app

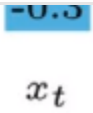
only important information is retained in s_t



Selective Write



Open in app

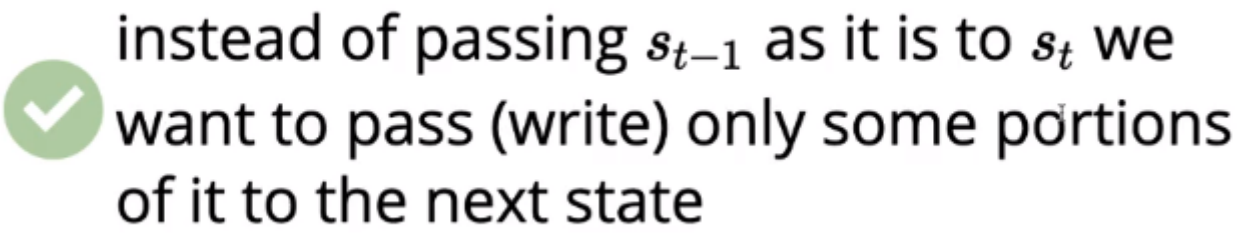


Equation of RNN:

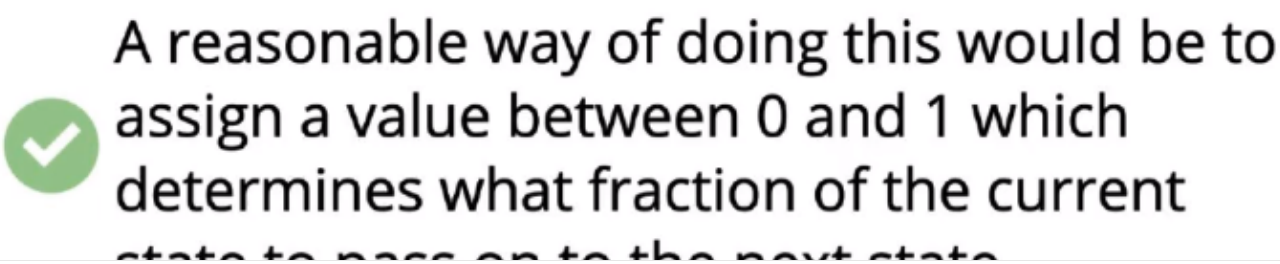
$$s_t = \sigma(Ux_t + W\mathbf{s_{t-1}} + b)$$

Bias(in the equation) is not shown in the figure because it does not depend on any of the input, for the sake of simplicity we can assume that bias b = 0.

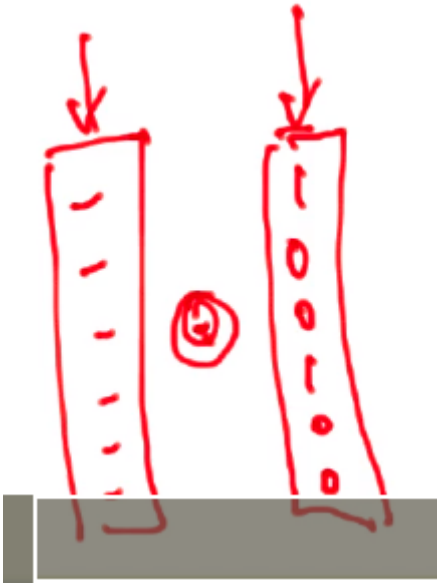
While computing **st**, instead of writing **s(t-1)** as it is and then using it, we could probably compress it a bit; we could write only some portion of it and use that to compute **st**.



Now **s(t-1)** is a 'd' dimension vector and we want only some entries from it to pass to **st**, one way of doing that would be that we multiply this **s(t-1)** vector with another vector which has the value 1 at the same index(the same index as that of **s(t-1)** entries which we want to pass to **st**) and all other values in this second vector would be 0. So, effectively we get only those entries from **st** for which the corresponding index is 1.

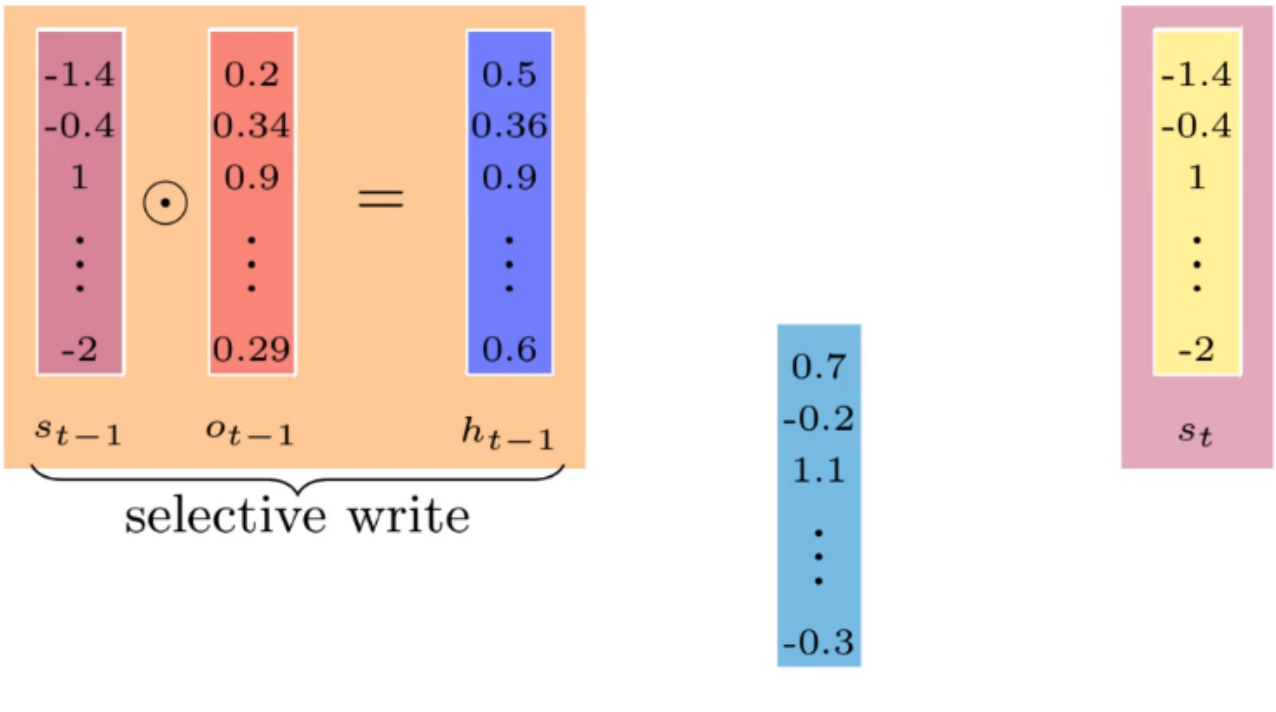


Open in app



Another way of doing this would be that we assign values to all the entries in the 'd' dimensional vector(**s(t-1)**) and these values would lie in the range (0–1). We could assign a very low value to those entries/index which are not important.

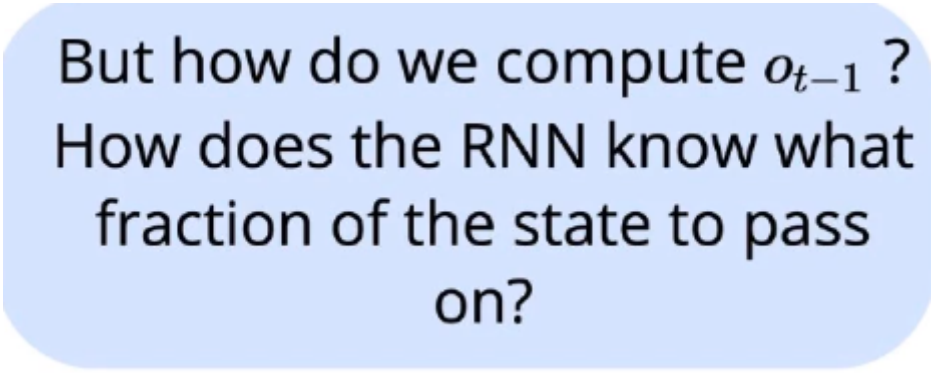
Every element of **s(t-1)**, we multiply it with a value between 0 to 1 that decides how important this value is. In the extreme case, there could be some numbers which are 0 here that means that value was not at all important and some values could be 1 which means that this value is very important and using this we compute an intermediate state **h(t-1)** which we then use to compute **st**.



Open in app

mentioned concept

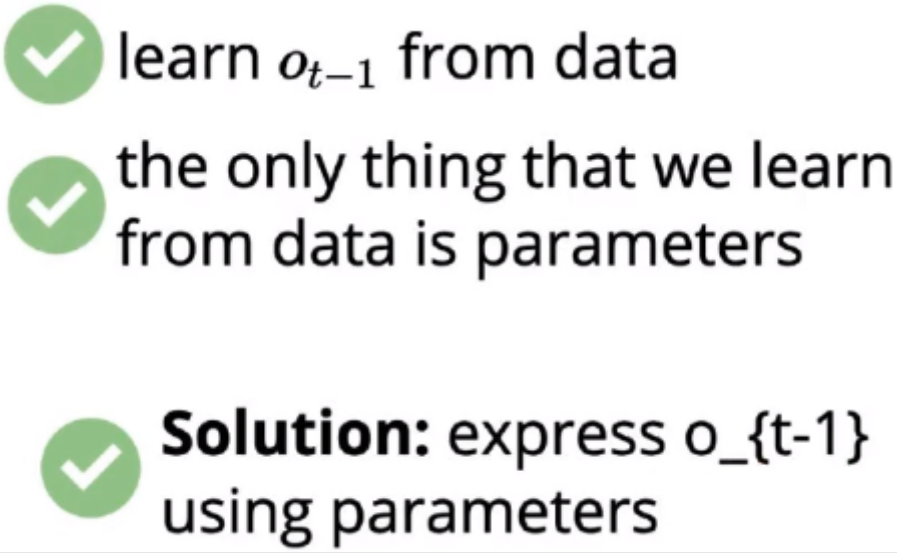
How do we compute o(t-1)?



The answer to the above question is that we learn the values of **o(t-1)** from data just like the way we learn the values of **U** and **W** from data in a way such that some loss is minimized.

o(t-1) is not different, its also just a collection of numbers which are getting multiplied with **s(t-1)**, we don't know what these numbers are so we should just learn it from the data just as we learn **U** and **W**.

But the way we are able to learn **U** and **W** is that both **U** and **W** are parameters and now we are able to use this parametric learning where we have a parameter and we update the value of the parameter as per Gradient Descent or the other algorithm being used because loss would depend upon parameter and then we can compute the derivative of loss with respect to parameter and update it.



Open in app

$$o_{t-1} = \sigma(U_o x_{t-1} + W_o h_{t-2} + b_o)$$
$$h_{t-1} = s_{t-1} \odot o_{t-1}$$

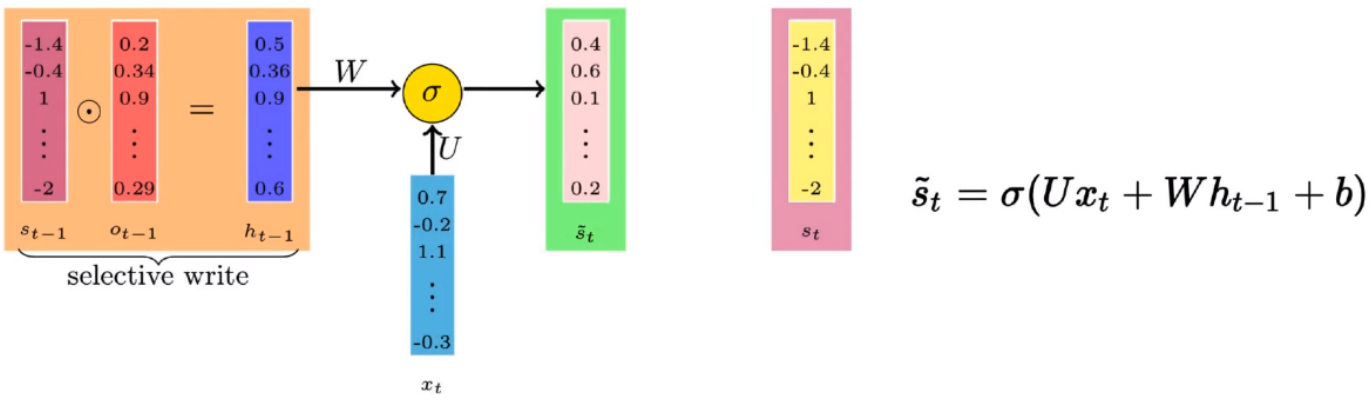
We initialize the value U0, W0, b0 to some random values because of which loss at very first computation might be high which would then be reduced in successive iterations as the parameters would get updated.

o_t is called the **output** gate

In LSTM terminology, **ot** is referred to as the output gate. It is termed as gate because it is gating the information, it is telling how much of the information to pass to the next state that's what a gate does, it decides who to pass, who not to pass, who should be allowed to pass and so on and that is what this o(t-1) is doing.

All the values in **o(t-1)** is between 0 to 1 as this vector **o(t-1)** is coming from a sigmoid function.

Selective Read

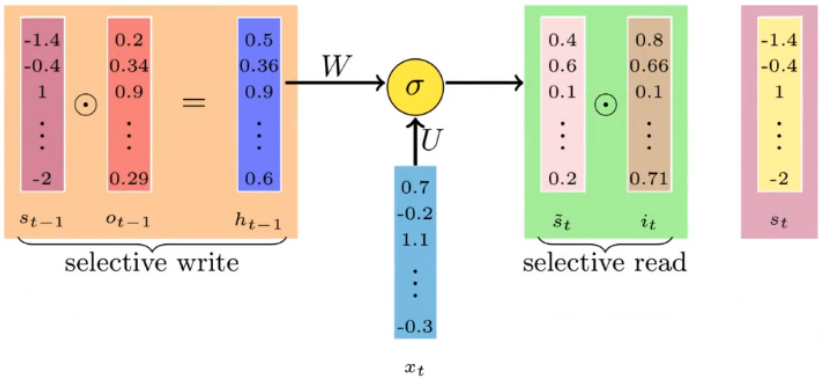


$$\tilde{s}_t = \sigma(U x_t + W h_{t-1} + b)$$

Open in app

from the previous state h_{t-1} and the current input x_t

However, we may not want to use all this new information and only selectively read from it before constructing the new cell state.



$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$
$$= \tilde{s}_t \odot i_t$$

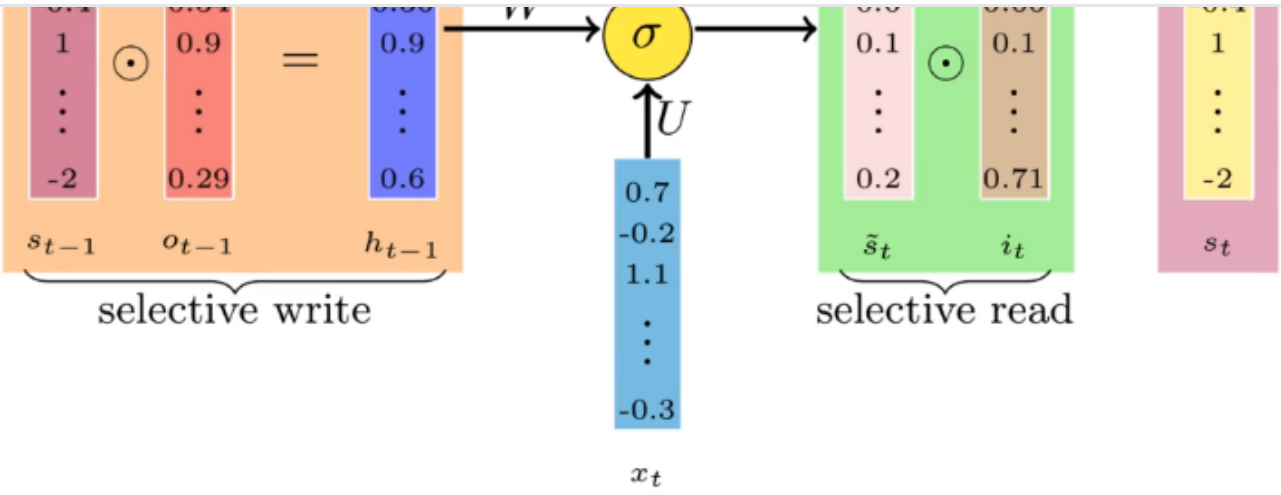
i_t is called the **input** gate

Input gate is a function of the current input, previous hidden state and some parameters Ui, Wi, bi and once again we learn these parameters from data in a way so that the loss function is minimized.

Initially Ui, Wi, bi would have some random values and hence '**it**' would have some random value as these parameters/weights would not have the right configuration initially. Hence, the loss function would also be very high. And once we compute the loss, we can update all these parameters and loss would become better at each iteration.

Till now we have covered the story/part till the selective read part in the image below:

Open in app



And all the computations/process till now is:

Previous state:

$$s_{t-1}$$

Output gate:

$$o_{t-1} = \sigma(W_o h_{t-2} + U_o x_{t-1} + b_o)$$

Selectively Write:

$$h_{t-1} = o_{t-1} \odot \sigma(s_{t-1})$$

Current (temporary) state:

$$\tilde{s}_t = \sigma(W h_{t-1} + U x_t + b)$$

Input gate:

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$

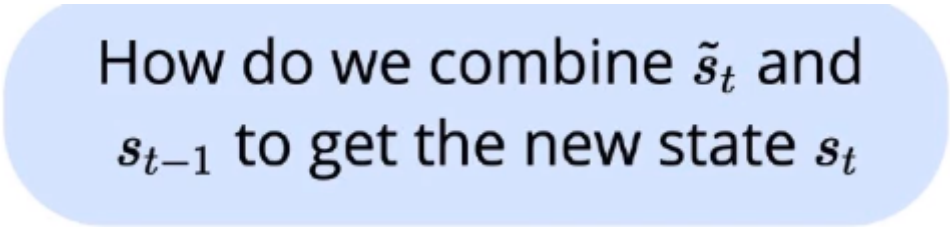
Selectively Read:

$$i_t \odot \tilde{s}_t$$

Selectively Forget

Open in app

property of selective write, read as discussed earlier.



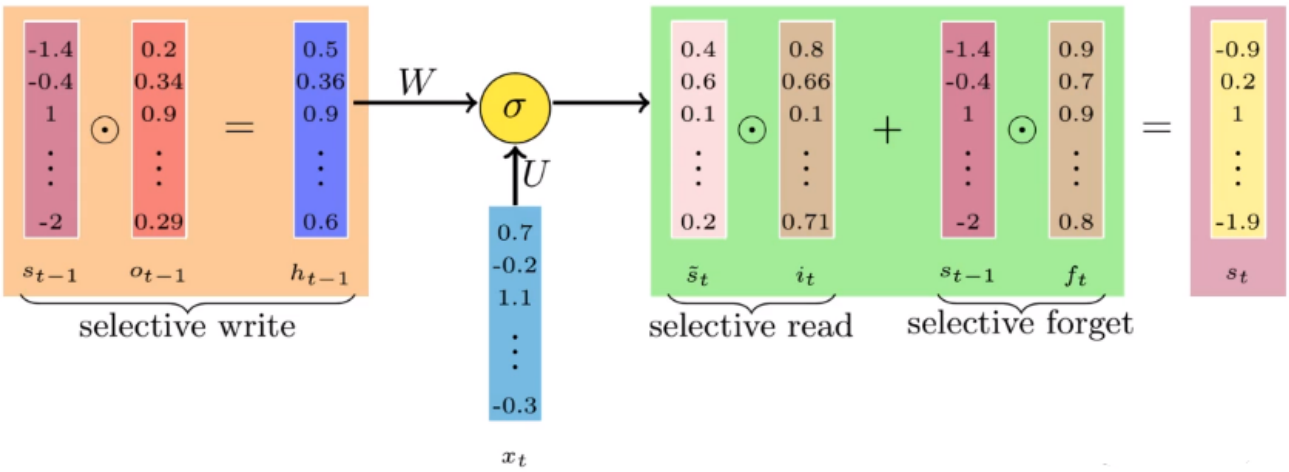
Now instead of using all the values from **s(t-1)**, we need to forget something and only retain the relevant information that's how they formulated it.

They came up with the below-mentioned equation:

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$
$$s_t = \tilde{s}_t \odot i_t + s_{t-1} \odot f_t$$

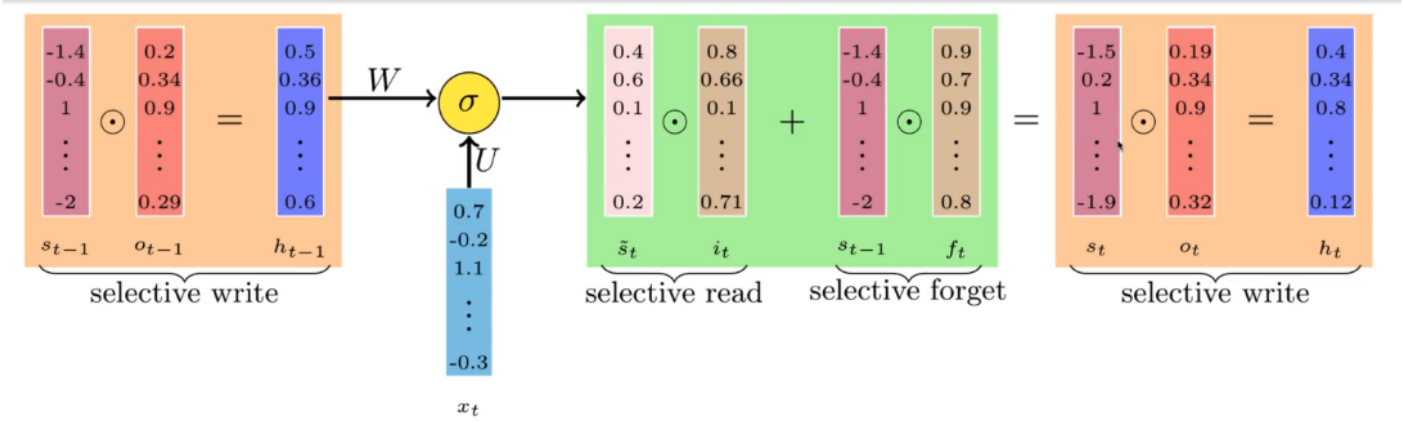
Forget Gate (**ft**) is again some values between 0 to 1. This decides what fraction of **s(t-1)** to retain in the final computation of **st**.

And this **ft** is again a standard recipe, its a function of some inputs which happens to be **xt** and previous intermediate state (**h(t-1)**) in this case and is also a function of some parameter which are **Uf**, **Wf**, **bf**, in this case also we learn these parameters from the data.



Open in app

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o) \qquad \tilde{s}_t = \sigma(W h_{t-1} + U x_t + b)$$
$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i) \qquad s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f) \qquad h_t = o_t \odot \sigma(s_t)$$



All the three gates depends upon **h(t-1)**, **xt**.

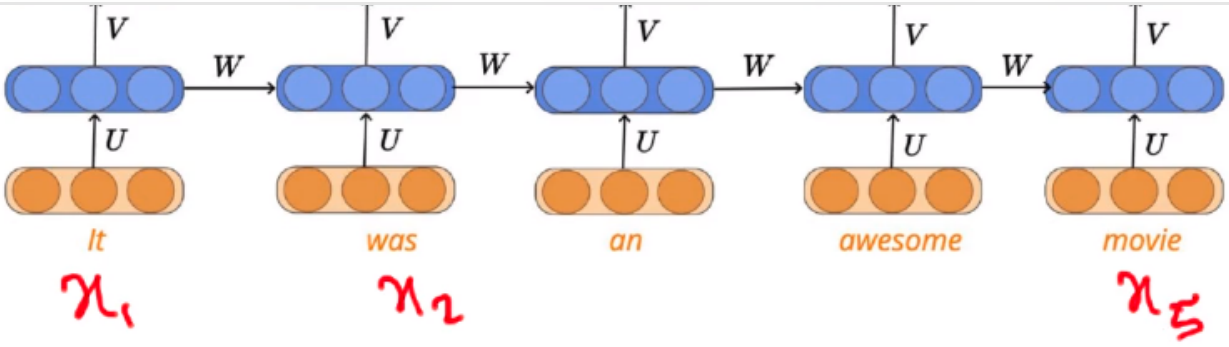
Earlier in the case of RNN, we had only 3 parameters i.e **W**, **U**, **b**.

Now in case of LSTM, we have **W**, **U**, **b** for **3 gates(with proper suffix)** in addition to the main **W**, **U**, **b** parameters making a total of 12 parameters(of which each in itself is a matrix). And all of these we are going to learn from the data such that the final loss is minimized. Training Algorithm would remain the same.

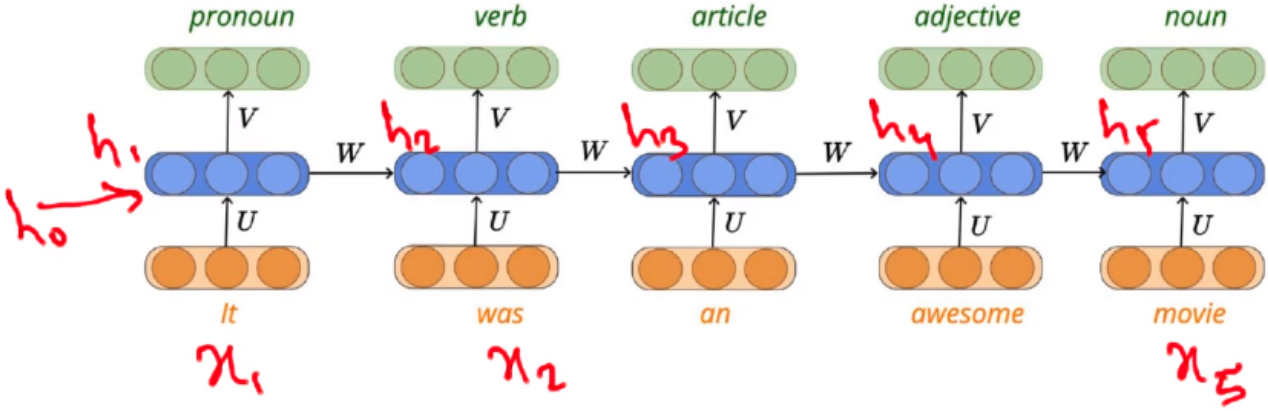
An Example Computation with LSTMs

Our input in this case is:

Open in app



State of the model at different time step is denoted by 'h' with proper suffix and is as per given in the image below:



Gates:

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

States:

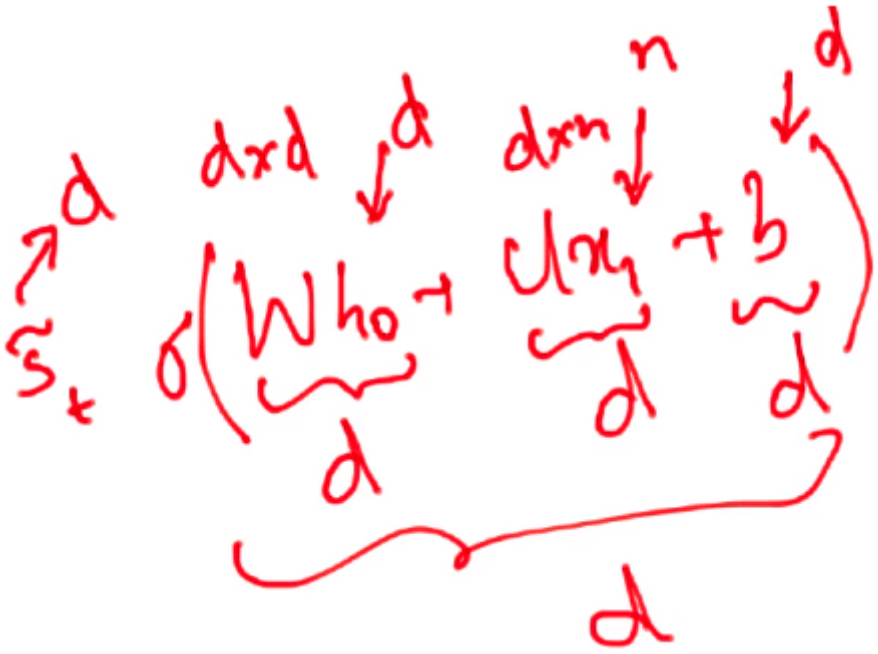
$$\tilde{s}_t = \sigma(W h_{t-1} + U x_t + b)$$
$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$$
$$h_t = o_t \odot \sigma(s_t)$$

The first thing we compute is **s1(~)** which would depend on **h0**, **x1**.

Assuming that **W**, **U**, **b** are given to us then this would be a simple computation.

h0 is going to be a '**d**' dimensional vector, **x1** is going to be a '**n**' dimensional vector, and **b** is going to be a '**d**' dimensional vector. We are assuming that **s1(~)** is going to be

Open in app



Next computation we would do is to calculate **st** which depends upon **ft**, **it**, **s0**(assuming that it is available), **st(~)**.

Now, this **f1** depends on the parameters **Wf**, **Uf**, **bf** and the vectors **h0**, **x1** which are already available and this looks very similar to the equation used to compute **s1(~)** except that the parameters are different.

Computing '**it**' is also the same as computing '**ft**' as both depend on the same inputs, the only thing different is the parameters.

So, we compute **s1** from all these values.

After this, we compute the **h1** using the values of **o1** and **s1**. **s1** is already computed and **o1** we can compute in a similar way as the **i1**.

And we keep computing the subsequent parameters like this and when we have the h5 output then we can compute **the final output as**

O(Vh5 + c)

where **O is the softmax function**.

Gated Recurrent Units:

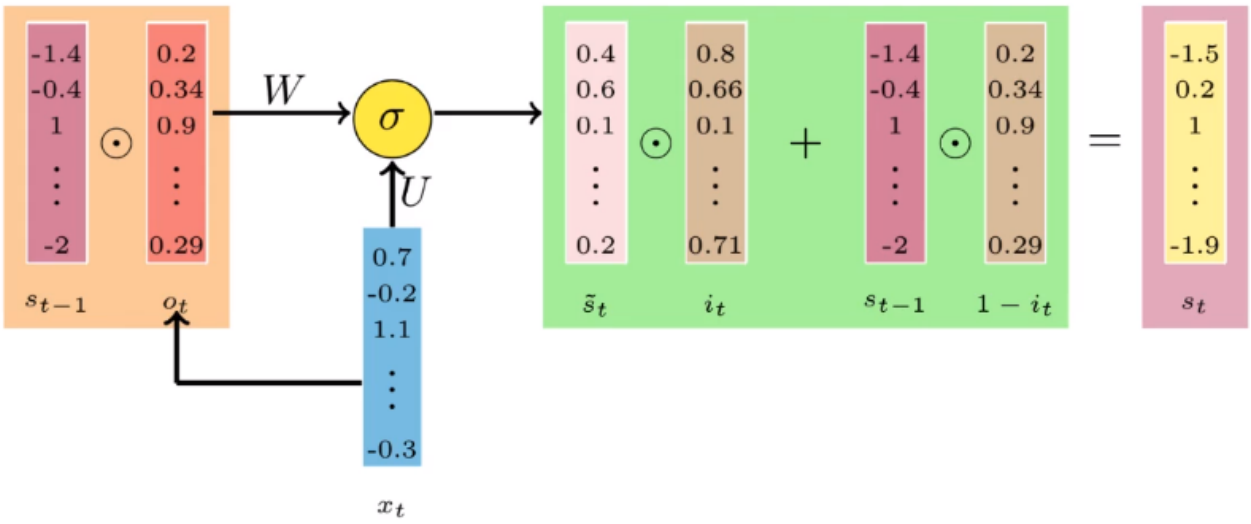
Open in app

- The one which we just saw is one of the most popular variants of LSTM
- Another equally popular variant of LSTM is Gated Recurrent Unit which we will see next

Gated Recurrent Unit is exactly the same as the LSTM except for one minor change and this change is **when we need to combine/sum up the st(~) and s(t-1)**, there instead of using the forget gate '**ft**' we use the value (**1 — it**) and the reason behind this is that since '**it**' **values lie in the range 0 to 1**, if we take (**it * st(~)**) that means **we are taking a fraction of st(~) then the remaining fraction should come from s(t-1)**, that's how we are going to combine this temporary state st(~) and the old state that we had (s(t-1)).

When we combine these two states, we decide how much of the current state to read which is '**it**' and then whatever fraction remains that we read from **s(t-1)**. So, in a way we have

ft = (1 — it)



Gates:

$$o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$$

States:

$$\tilde{s_t} = \sigma(W(o_t \odot s_{t-1}) + U x_t + b)$$
$$s_t = (1 - i_t) \odot s_{t-1} + i_t \odot \tilde{s_t}$$

Open in app

Earlier we were explicitly computing **ht**, but now we are not doing that we are just directly using its value as it is(red box in the image below), so we are not maintaining that additional state, we are directly using it and the final state is again referred to as '**st**'.

In case of LSTMs we are computing both '**st**' and '**ht**' but here we are directly computing '**st**' only.

States:

$$\tilde{s}_t = \sigma(W(o_t \odot s_{t-1}) + Ux_t + b)$$

$$s_t = (1 - i_t) \odot s_{t-1} + i_t \odot \tilde{s}_t$$

Summary:

So, we had this finite-state vector '**st**', and at every time step we were overriding it with new information, then we looked at in the light of the whiteboard analogy, we realized that we are overwhelming this vector a lot as at every time step we were writing something new and hence we wanted to do this selectively read, write and forget just as we do in a whiteboard which has a fixed size. Earlier, going from **s(t-1)** was easy because it only required the multiplication with **W** and then addition with (**U***xi) but now we have introduced this concept of gates and gates decide what fraction of information from original information we should selectively write, selectively read from and selectively forget. And now going from current state(which is **h(t-1)**) to a new state which is **ht** is a much more involved process because we have to do multiple computations, first we have a temporary state, then we need to compute all these 3 different gates, then we compute **st** from that we compute the final output **ht** and while doing this we are using all the gates that were involved.

Then we looked at the question, Is this always required? Do we always need three gates? So, for that, we saw the GRUs and how it combines the forget gate and the input gate. The idea is very simple there that terms that make the final state **st**, their total contribution should be 1(or to say 100%) so whatever be the contribution of one of

Open in app

computing this **ht**.

There are many other variants of LSTMs which uses combinations of Gates.

All the images used in this article is taken from the content covered in the LSTM and GRU module of the Deep Learning Course on the site: padhai.onefourthlabs.in

Machine Learning Deep Learning Recurrent Neural Network Rnn Padhai

About Write Help Legal

Get the Medium app

Download on the App Store GET IT ON Google Play